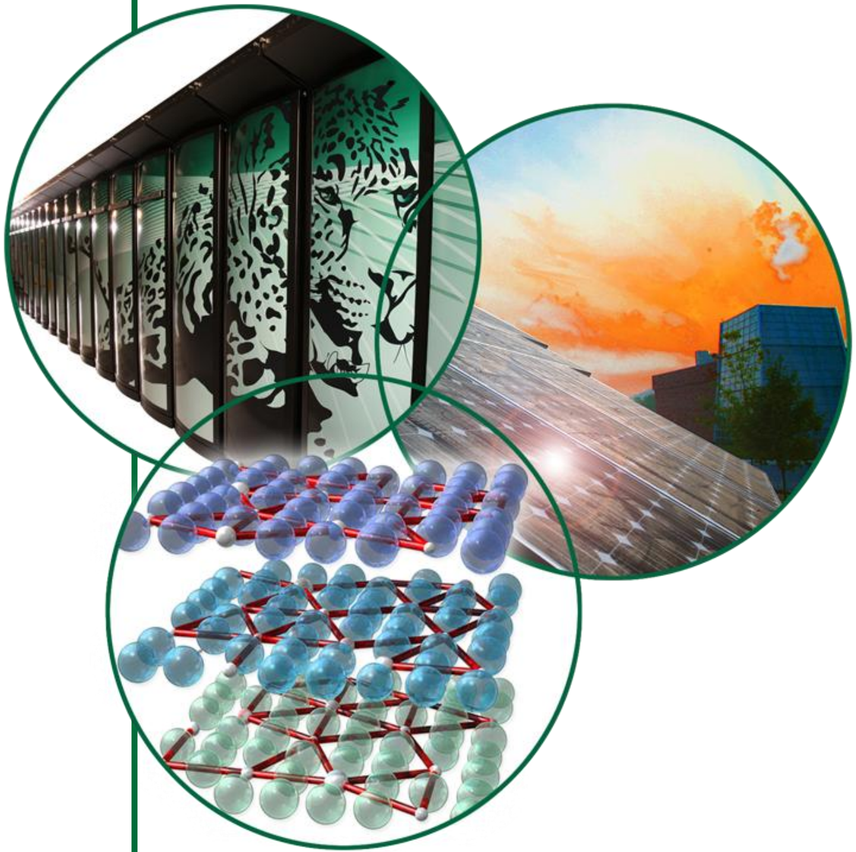
Vampir Introduction

Trace-based Performance Analysis

Thomas Ilsche, ORNL

May 17, 2011



U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Vampir – Tool Suite

- Help you with optimizing your parallel application
- Provides a view into the execution of the application
- More detailed view than profiling
  (temporal and spatial dependencies etc.)
- Vampir does not fix your code
  Vampir does not optimize your code
  You are the expert – you draw the conclusions

Motivation

- Why performance analysis?
  - Efficient usage of limited resources
  - Increase scalability for bigger simulations

- Profiling and Tracing
  - Include optimization as a phase in your development
  - Use tools instead of printf-solutions

Profiling and Tracing

- Instrumentation
  - Detect events (points of interest) during execution
  - Handle that information in a measurement library
- Profiling
  - Aggregates the available information
  - Count the time spent in a function and sum it up
- Trace recording
  - Save the individual event with a timestamp and processes/thread information

Profiling and Tracing

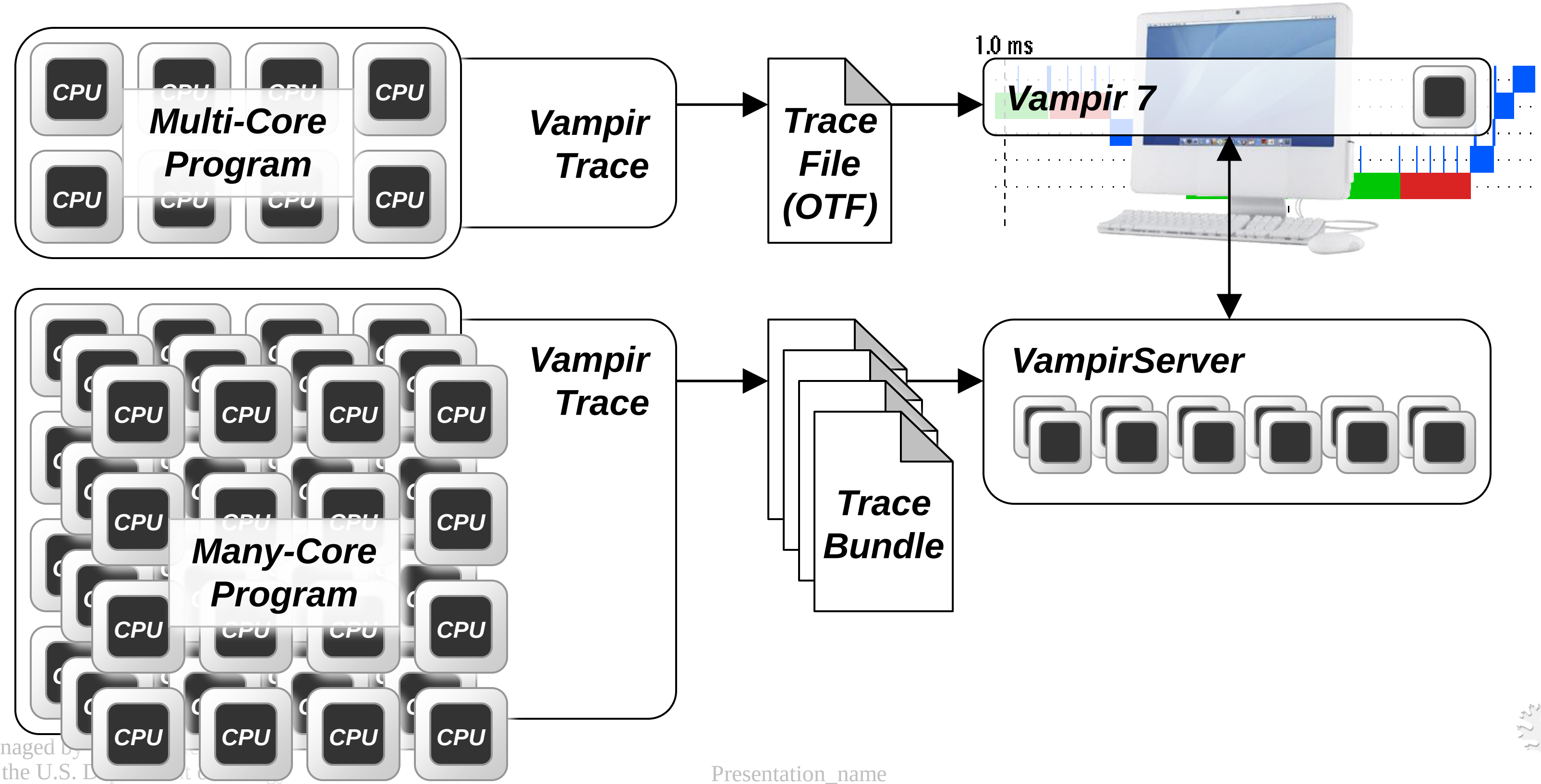
- Tracing advantages
  - Preserve the temporal and spatial relationships of events
  - Profiles can be calculated from a trace but not vice versa
- Tracing disadvantages
  - Traces can become very large
  - More perturbation than just profiling
  - Instrumentation and tracing is more complex
    - Larger I/O
    - Event buffering
    - Clock synchronization issues

Common Event Types

- All Types have timestamp and process information
- Enter and leave of a function/region
  - Region ID
- Send and receive of messages (MPI, GPU<->Host)
  - Sender, receiver, size, tag, communicator
- Collective communication (MPI)
  - Root, communicator, (size)
- Performance counter values (PAPI)
  - Counter ID, value

Vampir – Tool Suite

- The Vampir Performance Analysis Suite consists of
  - VampirTrace: Collect trace data
  - Vampir: The Graphical User Interface for trace analysis
  - VampirServer: A parallel performance analysis engine



VampirTrace

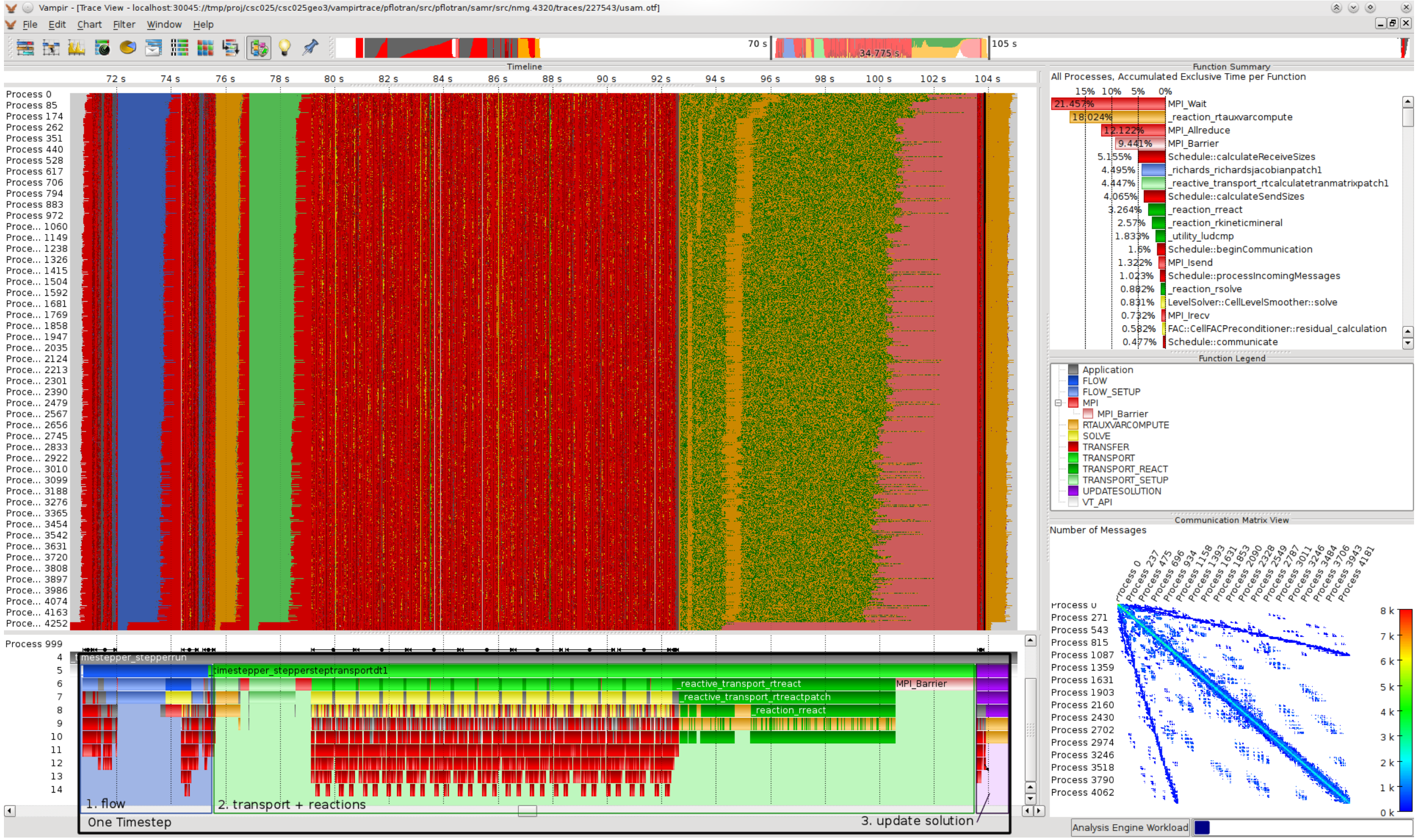
- VampirTrace consists of
  - Trace library
  - Compiler wrapper
  - Tools to process trace files
- VampirTrace collects timestamped events
  - No aggregation of data (by default)
  - All information is preserved for analysis
  - Trace files can become large and hard to handle
  - VampirTrace uses two file handles / process, which is a difficult for LUSTRE to handle on large applications
    - Work in progress to fix this

VampirTrace

- VampirTrace supports the tracing of
  - Function calls, using
    - Compiler instrumentation
    - Manual instrumentation (regions)
    - Binary instrumentation (dyninst)
    - Wrapping library calls
  - MPI
    - Point to point
    - Collectives
    - I/O
  - Hardware Counters (PAPI)
  - CUDA events
    - Memory copy
    - Kernel execution

Vampir

- GUI to analyze trace files (OTF)
- Main concept: Timeline
- GUI is QT based – available on Linux, Mac, Windows



Vampir - [Trace View - localhost:30045://tmp/proj/csc025/csc025geo3/vampirtrace/pflotran/src/pflotran/samr/src/nmg.4320/traces/227543/usam.otf]

Function Summary

All Processes, Accumulated Exclusive Time per Function

21.457%	MPI_Wait
18.024%	_reaction_rtauxvarcompute
12.122%	MPI_Allreduce
9.441%	MPI_Barrier
5.155%	Schedule::calculateReceiveSizes
4.495%	_richards_richardsjacobianpatch1
4.447%	_reactive_transport_rtcalculatetranmatrixpatch1
4.065%	Schedule::calculateSendSizes
3.264%	_reaction_rreact
2.57%	_reaction_rkineticmineral
1.833%	_utility_ludcmp
1.6%	Schedule::beginCommunication
1.322%	MPI_Isend
1.023%	Schedule::processIncomingMessages
0.882%	_reaction_rsolve
0.831%	LevelSolver::CellLevelSmoother::solve
0.732%	MPI_Irecv
0.582%	FAC::CellFACPreconditioner::residual_calculation
0.477%	Schedule::communicate

VampirServer

- Parallel analysis engine for Vampir
  - MPI
  - OpenMP
- Scales to > 10,000 analysis processes
- Loads the entire uncompressed trace into memory

Finding Performance Bottlenecks

- Inefficient Communication patterns
- Load imbalance / serial parts of the application
- Memory bound computation
  - Inefficient cache usage
  - TLB misses
  - Use HW counters (PAPI) to detect
- I/O bottlenecks
- Most time consuming function

Effects due to Tracing

- I/O overhead (flush)
  - Visibly marked in the trace
  - ‘Long‘ time for I/O
  - Ideally only once at the end (invisible) or during barriers
  - Avoid by applying runtime filters
- Measurement overhead
  - Overhead on function calls
  - Invisible
  - Avoid instrumenting tiny frequently called functions
  - Compare total runtime to get an upper bound on overhead

Conclusion

- Performance analysis is very important in HPC
- Use the right tool for your needs
- Use tracing with caution
- Contact me for questions, problems or feature wishes

Thank you

- Contact:
  - Thomas Ilsche, ORNL
    5700 B206
    tt1@ornl.gov (`Thomas.Ilsche@zih.tu-dresden.de`)
    865-241-6293